July, 2003

An extension of HM(X) with bounded existential and universal data-types

(To appear at ICFP'03)

Vincent Simonet

INRIA Rocquencourt — Cristal project

`Vincent.Simonet@inria.fr`
`http://cristal.inria.fr/~simonet/`

INRIA
ROCQUENCOURT

The initial problem

Flow Caml is an extension of the Objective Caml language with a type system tracing information flow. Usual ML types are annotated by security levels, which represent principals (e.g. human beings `!alice`, `!bob`, ...). A partial order between these levels specifies legal information flow, hence the type system has subtyping.

```
type ('a:level) client_info =
  { cash: 'a int;
    send_msg: 'a int -> unit;
    ...
  }
```

Problem: the types `!alice client_info` and `!bob client_info` are not comparable.

ML with First-Class Abstract Types

Odersky and Läufer proposed an extension of ML where data-types declarations may introduce existentially quantified variables:

```
type t = K of Exists 'a . 'a list * ('a -> unit)
```

This extension preserves type inference: the annotation provided by the introduction and the matching of the constructor `K` are sufficient to guide the type synthetizer.

```
let v1 = K ([3; 42; 111], print_int)
let v2 = K (["Hello"; "World"], print_string)
let iter = function K (x, f) -> List.iter f x
```

Existential type variables cannot escape their scope. The following piece of code is ill-typed:

```
let open = function K (x, _) -> x
```

ML with First-Class Polymorphic Types

Symmetrically, universally quantified type variables can be introduced in data-types declarations [Rémy, 1994]:

```
type t = L of ForAll 'a . ('a list -> 'a)
```

They are in particular useful in presence of abstract data-types:

```
let apply g = function K (x, f) -> f (g x)
```

is ill typed, but one can write:

```
let apply (L g) = function K (x, f) -> f (g x)
```

(Poor man's first class polymorphism)

Our work

$\mathrm{HM}(X)$ is a generic constraint-based type inference system with let-polymorphism. It generalizes Hindley-Milner type system.

It is parametrized by the first-order logic X, which is used to express types and constraints relating them. The type inference problem is reduced to solving constraints in the logic.

- We define a conservative extension of $\mathrm{HM}(X)$ with bounded existential and universal data-types.
- We propose a realistic algorithm for solving constraints in the case of structural subtyping.

The type system

Types and constraints

We assume two distinct sets of existential ε and universal π type constructors.

$$\begin{array}{rcll} \tau & ::= & \alpha, \beta, \dots \mid \tau \to \tau \mid \varepsilon(\bar{\tau}) \mid \pi(\bar{\tau}) & \text{(type)} \\ C, D & ::= & \tau \leq \tau \mid C \wedge C \mid \exists \alpha . C & \text{(constraint)} \\ \sigma & ::= & \forall \bar{\alpha}[C].\tau & \text{(scheme)} \end{array}$$

Every data-type must be introduced by a declaration:

$$\varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau \qquad \pi(\bar{\alpha}) \triangleq \forall \bar{\beta}[D].\tau$$

Subtyping

The interpretation of the subtyping order between types is left open. However, →, ε and π types must be incomparable and the variances of the existential and universal type constructors must fit their logical interpretation:

$$\varepsilon(\bar\alpha_1) \triangleq \exists\bar\beta_1[D_1].\tau_1 \qquad \varepsilon(\bar\alpha_2) \triangleq \exists\bar\beta_2[D_2].\tau_2$$

with $\bar\beta_2 \# \mathrm{fv}(\tau_1)$ imply

$$D_1 \wedge \varepsilon(\bar\alpha_1) \leq \varepsilon(\bar\alpha_2) \vDash \exists\bar\beta_2.(D_2 \wedge \tau_1 \leq \tau_2)$$

$$\pi(\bar\alpha_1) \triangleq \forall\bar\beta_1[D_1].\tau_1 \qquad \pi(\bar\alpha_2) \triangleq \forall\bar\beta_2[D_2].\tau_2$$

with $\bar\beta_1 \# \mathrm{fv}(\tau_2)$ imply

$$D_2 \wedge \pi(\bar\alpha_1) \leq \pi(\bar\alpha_2) \vDash \exists\bar\beta_1.(D_1 \wedge \tau_1 \leq \tau_2)$$

Several instances: unification, non-structural subtyping, structural subtyping.

The language

We extend the λ-calculus with explicit constructs for packing and opening existential and universal values:

$$\begin{array}{rcll} e & ::= & x \mid \lambda x.e \mid e\,e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e & \text{(expression)} \\ & \mid & \langle e \rangle_\varepsilon \mid \mathsf{open}_\varepsilon\, e\, \mathsf{with}\, e & \\ & \mid & \langle e \rangle_\pi \mid \mathsf{open}_\pi\, e & \end{array}$$

The (call-by-value) semantics is extended as follows:

$$\begin{array}{rcll} \mathsf{open}_\varepsilon\, \langle v \rangle_\varepsilon\, \mathsf{with}\, (\lambda x.e) & \longrightarrow & (\lambda x.e)\, v & (\varepsilon) \\ \mathsf{open}_\pi\, \langle v \rangle_\pi & \longrightarrow & v & (\pi) \end{array}$$

Standard HM(X) typing rules

$$\textsc{Var}\quad \frac{\Gamma(x) = \forall\bar{\alpha}[D].\tau \qquad C \vDash D}{C, \Gamma \vdash x : \tau}$$

$$\textsc{Abs}\quad \frac{C, \Gamma[x \mapsto \tau'] \vdash e : \tau}{C, \Gamma \vdash \lambda x.e : \tau' \to \tau}$$

$$\textsc{App}\quad \frac{C, \Gamma \vdash e_1 : \tau' \to \tau \quad C, \Gamma \vdash e_2 : \tau'}{C, \Gamma \vdash e_1\, e_2 : \tau}$$

$$\textsc{Let}\quad \frac{C, \Gamma \vdash e_1 : \sigma \quad C, \Gamma[x \mapsto \sigma] \vdash e_2 : \tau}{C, \Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}$$

$$\textsc{Generalize}\quad \frac{C \wedge D, \Gamma \vdash e : \tau \quad \bar{\alpha} \# \mathrm{fv}(C, \Gamma)}{C \wedge \exists\bar{\alpha}.D, \Gamma \vdash e : \forall\bar{\alpha}[D].\tau}$$

$$\textsc{Sub}\quad \frac{C, \Gamma \vdash e : \tau' \quad C \vDash \tau' \leq \tau}{C, \Gamma \vdash e : \tau}$$

$$\textsc{Hide}\quad \frac{C, \Gamma \vdash e : \tau \quad \bar{\alpha} \# \mathrm{fv}(\Gamma, \tau)}{\exists\bar{\alpha}.C, \Gamma \vdash e : \tau}$$

Typing rules for the new constructs

$$\textsc{Exist}\quad \frac{C, \Gamma \vdash e : \tau \qquad \varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau \qquad C \vDash D}{C, \Gamma \vdash \langle e \rangle_{\varepsilon} : \varepsilon(\bar{\alpha})}$$

$$\textsc{OpenExist}\quad \frac{\varepsilon(\bar{\alpha}) \triangleq \exists \bar{\beta}[D].\tau' \qquad \begin{array}{c} C, \Gamma \vdash e_1 : \varepsilon(\bar{\alpha}) \\ C, \Gamma \vdash e_2 : \forall \bar{\beta}[D].\tau' \rightarrow \tau \end{array} \qquad \bar{\beta} \# \mathrm{fv}(\tau)}{C, \Gamma \vdash \mathsf{open}_{\varepsilon}\, e_1 \,\mathsf{with}\, e_2 : \tau}$$

$$\textsc{Poly}\quad \frac{C, \Gamma \vdash e : \forall \bar{\beta}[D].\tau \qquad \pi(\bar{\alpha}) \triangleq \forall \bar{\beta}[D].\tau}{C, \Gamma \vdash \langle e \rangle_{\pi} : \pi(\bar{\alpha})}$$

$$\textsc{OpenPoly}\quad \frac{C, \Gamma \vdash e : \pi(\bar{\alpha}) \qquad \pi(\bar{\alpha}) \triangleq \forall \bar{\beta}[D].\tau \qquad C \vDash D}{C, \Gamma \vdash e : \tau}$$

Type safety

An expression e is well-typed if $C, \varnothing \vdash e : \tau$ holds for some satisfiable constraint C.

The type system has standard subject-reduction and progress theorems.

“Well-typed expressions do not go wrong”

Generating constraints

Outline

We define an algorithm for computing principal typing judgments:

$$(\!|\Gamma \vdash e : \tau|\!) \;\rightsquigarrow\; C$$

The algorithm must be correct: for all Γ, e and τ,

$$(\!|\Gamma \vdash e : \tau|\!), \Gamma \vdash e : \tau$$

and complete: for all C, Γ, e and τ,

$$\text{if } C, \Gamma \vdash e : \tau \text{ then } C \vDash (\!|\Gamma \vdash e : \tau|\!).$$

Core language

$$(\!|\Gamma \vdash x : \tau|\!) = \exists\bar{\alpha}.(C \wedge \tau' \leq \tau)$$

where $\Gamma(x) = \forall\bar{\alpha}[C].\tau'$

$$(\!|\Gamma \vdash \lambda x.e : \tau|\!) = \exists\alpha_1\alpha_2.((\!|\Gamma[x \mapsto \alpha_1] \vdash e : \alpha_2|\!) \wedge \alpha_1 \rightarrow \alpha_2 \leq \tau)$$

$$(\!|\Gamma \vdash e_1\, e_2 : \tau|\!) = \exists\alpha.((\!|\Gamma \vdash e_1 : \alpha \rightarrow \tau|\!) \wedge (\!|\Gamma \vdash e_2 : \alpha|\!))$$

$$(\!|\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau|\!) = (\!|\Gamma[x \mapsto \forall\alpha[C].\alpha] \vdash e_2 : \tau|\!) \wedge \exists\alpha.C$$

where $C = (\!|\Gamma \vdash e_1 : \alpha|\!)$

Existential and universal data-types

We introduce a non-standard construct in constraints:

$\forall\bar\beta.D \rhd C$ interpreted as “ $\exists\bar\beta.D \wedge \forall\bar\beta D \Rightarrow C$ ”

$$\llparenthesis \Gamma \vdash \langle e\rangle_\varepsilon : \tau \rrparenthesis = \exists\bar\alpha.(\exists\bar\beta.(\llparenthesis \Gamma \vdash e : \tau' \rrparenthesis \wedge D) \wedge \varepsilon(\bar\alpha) \le \tau)$$

$$\llparenthesis \Gamma \vdash \mathsf{open}_\varepsilon\, e_1\, \mathsf{with}\, e_2 : \tau \rrparenthesis = \exists\bar\alpha.(\llparenthesis \Gamma \vdash e_1 : \varepsilon(\bar\alpha) \rrparenthesis \wedge \forall\bar\beta.D \rhd \llparenthesis \Gamma \vdash e_2 : \tau' \to \tau \rrparenthesis)$$

where $\varepsilon(\bar\alpha) \triangleq \exists\bar\beta[D].\tau'$

$$\llparenthesis \Gamma \vdash \mathsf{open}_\pi\, e : \tau \rrparenthesis = \exists\bar\alpha.(\llparenthesis \Gamma \vdash e : \pi(\bar\alpha) \rrparenthesis \wedge \exists\bar\beta.(D \wedge \tau' \le \tau))$$

$$\llparenthesis \Gamma \vdash \langle e\rangle_\pi : \tau \rrparenthesis = \exists\bar\alpha.(\forall\bar\beta.D \rhd \llparenthesis \Gamma \vdash e : \tau' \rrparenthesis \wedge \pi(\bar\alpha) \le \tau)$$

where $\pi(\bar\alpha) \triangleq \forall\bar\beta[D].\tau'$

Summary

An expression e is well-typed in $\mathrm{HM}_{\exists\forall}(X)$ in and only if the constraint $\exists\alpha.(\!|\varnothing \vdash e : \alpha|\!)$ is satisfiable in the logic X. This constraint belongs to the following language:

$$C, D \quad ::= \quad \tau \leq \tau \mid C \wedge C \mid \exists\alpha.C \mid \forall\bar{\beta}.D \triangleright C$$

where every bound $\bar{\beta}.D$ of a universal quantification comes from a data-type declaration.

It remains to provide algorithms that solve these constraints.

Solving constraints: The case of structural subtyping

Overview

We need an algorithm for solving constraints which include a restricted form of universal quantification and implication.

On the one-hand, efficient (polynomial) algorithms that decide top-level implication of constraints ($C_1 \vDash C_2$, where all free variables are implicitely universally quantified) are known.

On the other hand, Kuncak and Rinard recently showed [LICS 2003] that the first order theory of structural subtyping is decidable, but their algorithm has a non-elementary complexity.

We strike a compromise between expressiveness and efficiency:

- thanks to the "weak" interpretation of $\forall\bar{\beta}.D \triangleright C$ which implies $\exists\bar{\beta}.D$,
- by restricting the form of the quantification bounds in every construct $\forall\bar{\beta}.D \triangleright C$.

A model of structural subtyping

Let a variance ν be one of $\oplus$ (covariant), $\ominus$ (contravariant) and $\odot$ (invariant).

We assume given a set of symbols φ. Every symbol has a fixed arity $a(\varphi)$ and a signature $\mathrm{sig}(\varphi) = [\nu_1, \ldots, \nu_{a(\varphi)}]$. Then ground types are defined by:

$$t \quad ::= \quad \varphi(t_1, \ldots, t_{a(\varphi)}) \quad \text{(ground type)}$$

Symbols of arity 0 are ground atoms: we suppose they are partially ordered by the lattice order $\leq_0$. Then, subtyping is defined by:

$$\frac{\varphi \leq_0 \varphi'}{\varphi \leq \varphi'} \qquad \frac{\mathrm{sig}(\varphi) = [\nu_1, \ldots, \nu_n] \qquad \forall i \;\; t_i \leq^{\nu_i} t'_i}{\varphi(t_1, \ldots, t_n) \leq \varphi(t'_1, \ldots, t'_n)}$$

$$\boxed{\begin{array}{l} \leq^{\oplus} \;\rightsquigarrow\; \leq \\ \leq^{\ominus} \;\rightsquigarrow\; \geq \\ \leq^{\odot} \;\rightsquigarrow\; = \end{array}}$$

Shapes

In structural subtyping, two comparable types must have the same shape. We define the relation $t \approx t'$ (read: t has the same shape as t') by:

$$\frac{}{\varphi \approx \varphi'} \qquad \frac{\mathrm{sig}(\varphi) = [\nu_1, \ldots, \nu_n] \qquad \forall i \;\; t_i \approx^{\nu_i} t'_i}{\varphi(t_1, \ldots, t_n) \approx \varphi(t'_1, \ldots, t'_n)}$$

$$\boxed{\begin{array}{l} \approx^{\oplus} \;\rightsquigarrow\; \approx \\ \approx^{\ominus} \;\rightsquigarrow\; \approx \\ \approx^{\odot} \;\rightsquigarrow\; = \end{array}}$$

$\approx$ is the reflexive, symmetric, transitive closure of $\leq$. Its equivalence classes are lattices.

Expansion and decomposition

In structural subtyping, the two following equivalence rules hold:

Expansion:
$$\begin{aligned} \varphi(\bar\tau) \leq \alpha &\equiv \exists\bar\alpha.(\varphi(\bar\alpha) = \alpha \wedge \varphi(\bar\tau) \leq \varphi(\bar\alpha)) \\ &\equiv \exists\langle\varphi(\bar\alpha) = \alpha\rangle.(\varphi(\bar\tau) \leq \varphi(\bar\alpha)) \end{aligned}$$

Decomposition: $\mathrm{sig}(\varphi) = [\nu_1, \ldots, \nu_n]$
$$\varphi(\tau_1, \ldots, \tau_n) \leq \varphi(\tau'_1, \ldots, \tau'_n) \equiv \tau_1 \leq^{\nu_1} \tau'_1 \wedge \cdots \wedge \tau_n \leq^{\nu_n} \tau'_n$$

Our algorithm consists in rewriting the input constraint into a solved form:

$$\begin{array}{lcll} \eta & ::= & \varphi \mid \alpha & \text{(atom)} \\ R & ::= & \varnothing \mid \eta \leq \eta \wedge R \mid \eta \approx \eta \wedge R & \text{(multiset of atomic constraints)} \\ S & ::= & R \mid \exists\langle\phi(\bar\alpha) = \alpha\rangle.S & \text{(solved form)} \end{array}$$

(In $\exists\langle\phi(\bar\alpha) = \alpha\rangle.S$, we require $\alpha \notin \mathrm{fv}(S)$).
By orienting the two above rules from left to right, we obtain an algorithm which rewrites any conjunction of inequalities into a solved form. It remains to eliminate quantifiers.

Eliminating existential quantifiers

Goal: $\exists\beta.S \rightsquigarrow S'$

$\exists\beta.[]$ commutes with $\exists\langle\phi(\bar{\alpha})=\alpha\rangle.[]$

$$\begin{array}{lcll}\exists\beta.\exists\langle\phi(\bar{\alpha})=\alpha\rangle.S & \rightsquigarrow & \exists\langle\phi(\bar{\alpha})=\alpha\rangle.\exists\beta.S & \text{if } \alpha\neq\beta \text{ (and } \beta\notin\bar{\alpha}\text{)}\\ \exists\alpha.\exists\langle\phi(\bar{\alpha})=\alpha\rangle.S & \rightsquigarrow & \exists\bar{\alpha}.S & \end{array}$$

$\exists\beta.[]$ can be eliminated when it reaches the multiset of atomic inequalities

$$\begin{array}{lcl}\exists\beta.R & \rightsquigarrow & \{\eta_1 \diamond \eta_2 \mid \eta_1 \diamond \eta_2 \in R \text{ and } \eta_1,\eta_2 \neq \beta\}\\ & & \cup\,\{\eta_1 \diamond_1\diamond_2 \eta_2 \mid \eta_1 \diamond_1 \beta \in R \text{ and } \beta \diamond_2 \eta_2 \in R\}\end{array}$$

where $\diamond$ ranges over $\approx$, $\leq$ and $\geq$.

$$\exists\beta.(\beta\leq\alpha_1\wedge\beta\leq\alpha_2) \rightsquigarrow \alpha_1\approx\alpha_2$$

Restricting universal quantification bounds

We consider a constraint $\forall\bar{\beta}.D \rhd C$.

- Existential quantifiers in D can be fused with the universal one:

$$\forall\bar{\beta}.(\exists\bar{\alpha}.D) \rhd C \;\equiv\; \forall\bar{\beta}\bar{\alpha}.D \rhd C$$

- Type constructors in D can be eliminated by expansion and decomposition, e.g.

$$\forall\beta.(\beta \leq \alpha_1 \rightarrow \alpha_2) \rhd C \;\equiv\; \forall\beta_1\beta_2.(\alpha_1 \leq \beta_1 \wedge \beta_2 \leq \alpha_2) \rhd C[\beta_1 \rightarrow \beta_2/\beta]$$

Thus, we may assume that D is a conjunction of inequalities involving atoms.

Restricting universal quantification bounds

Consider a constraint $\forall\bar{\beta}.D \rhd C$ and a variable $\beta \in \bar{\beta}$. Three situations may arise:

- β has no external bound in D, i.e. is only related to variables of $\bar{\beta}$. In this case, C cannot constrain its shape.
  For instance $\forall\beta.\mathbf{true} \rhd \beta \leq \alpha_1 \rightarrow \alpha_2$ is not satifiable.
- β has one lower and/or upper bound(s) in D.

$$\begin{array}{rl} & \forall\beta.(\beta \leq \alpha) \rhd (\beta \leq \alpha'_1 \rightarrow \alpha'_2) \\ \equiv & \exists\langle\alpha_1 \rightarrow \alpha_2 = \alpha\rangle.(\forall\beta.(\beta \leq \alpha_1 \rightarrow \alpha_2) \rhd (\beta \leq \alpha'_1 \rightarrow \alpha'_2)) \\ \equiv & \exists\langle\alpha_1 \rightarrow \alpha_2 = \alpha\rangle.(\forall\beta_1\beta_2.(\alpha_1 \leq \beta_1 \wedge \beta_2 \leq \alpha_2) \rhd (\alpha'_1 \leq \beta_1 \wedge \beta_2 \leq \alpha'_2)) \\ \equiv & \exists\langle\alpha_1 \rightarrow \alpha_2 = \alpha\rangle.(\alpha'_1 \leq \alpha_1 \wedge \alpha_2 \leq \alpha'_2) \end{array}$$

[...]

Restricting universal quantification bounds

[...]

- β has several lower or upper bounds in D.

$$\begin{aligned} & \forall\beta.(\beta \leq \alpha_1 \wedge \beta \leq \alpha_2) \rhd (\beta \leq \alpha) \\ \equiv \quad & \forall\beta.(\beta \leq \alpha_1 \sqcap \alpha_2) \rhd (\beta \leq \alpha) \\ \equiv \quad & \alpha_1 \sqcap \alpha_2 \leq \alpha \end{aligned}$$

We exclude this third case.

Some examples of allowed quantification bounds:

$$\begin{array}{ll} (1) & \forall\beta_1\beta_2\beta_3.(\beta_1 \leq \beta_2 \leq \beta_3) \rhd \cdots \\ (2) & \forall\beta_1\beta_2.(\alpha_1 \leq \beta_1 \leq \alpha_2 \wedge \alpha_1 \leq \beta_2 \leq \alpha_2) \rhd \cdots \\ (3) & \forall\beta_1\beta_2.(\varphi_1 \leq \beta_1 \leq \beta_2 \leq \varphi_2) \rhd \cdots \end{array}$$

Eliminating universal quantifiers

Goal: $\forall\bar\beta.D \rhd S \leadsto S'$

$\forall\bar\beta.D \rhd []$ commutes with $\exists\langle\phi(\bar\alpha)=\alpha\rangle.[]$

$$\begin{array}{rll}
\forall\bar\beta.D \rhd (\exists\langle\phi(\bar\alpha)=\alpha\rangle.S) & \leadsto \exists\langle\phi(\bar\alpha)=\alpha\rangle.(\forall\bar\beta.D[\phi(\bar\alpha)/\alpha] \rhd S) & \alpha \notin \bar\beta \\
\forall\alpha\bar\beta.D \rhd (\exists\langle\phi(\bar\alpha)=\alpha\rangle.S) & \leadsto \forall\bar\alpha\bar\beta.D[\phi(\bar\alpha)/\alpha] \rhd S & \alpha \text{ bounded} \\
\forall\alpha\bar\beta.D \rhd (\exists\langle\phi(\bar\alpha)=\alpha\rangle.S) & \leadsto \mathbf{failure} & \alpha \text{ unbounded}
\end{array}$$

$\forall\bar\beta.D \rhd []$ can be eliminated when it reaches the multiset

$$\begin{array}{rcl}
\forall\bar\beta.D \rhd R & \longrightarrow & (\exists\bar\beta.D) \\
& & \cup\,\{\mathrm{ub}_{\bar\beta.D}(\eta_1) \leq \mathrm{lb}_{\bar\beta.D}(\eta_2) \mid \eta_1 \leq \eta_2 \in R\backslash D^*\} \\
& & \cup\,\{\mathrm{sh}_{\bar\beta.D}(\eta_1) \approx \mathrm{sh}_{\bar\beta.D}(\eta_2) \mid \eta_1 \approx \eta_2 \in R\backslash D^*\}
\end{array}$$

$\mathrm{ub}_{\bar\beta.D}(\eta)$ is the upper bound of η under $\forall\bar\beta.D \rhd \cdots$

$\mathrm{lb}_{\bar\beta.D}(\eta)$ is the lower bound of η under $\forall\bar\beta.D \rhd \cdots$

$\mathrm{sh}_{\bar\beta.D}(\eta)$ is the shape of η under $\forall\bar\beta.D \rhd \cdots$

Summary

Our algorithm rewrites an arbitrary constraint into a solved form.

$$C \rightsquigarrow S$$

A solved form is satisfiable if and only if its multiset is satisfiable.

$$\begin{array}{lcll}
\eta & ::= & \varphi \mid \alpha & \text{(atom)} \\
R & ::= & \varnothing \mid \eta \leq \eta \wedge R \mid \eta \approx \eta \wedge R & \text{(multiset of atomic constraints)} \\
S & ::= & R \mid \exists \langle \phi(\bar{\alpha}) = \alpha \rangle . S & \text{(solved form)}
\end{array}$$

Examples

The bank example

In the lattice of security levels, we have one security level for every client (`!alice`, `!bob`, ...). We let `!clients` be their least upper bound.

```
type client_info = Exists 'a with 'a < !clients .
  { cash: 'a int;
    send_msg: 'a int -> unit;
    ...
  }
```

The bank example (2)

The function `send_balances` iterates over a list of clients and sends to each of them a message indicating their current balance:

```
let rec send_balances = function
  [] -> []
| { cash = x; send_msg = send } :: tl ->
    send x; send_balances tl
```

De-sugaring this example in the syntax of the current talk, we realize that the function which corresponds to the second case of the pattern matching

$$\lambda x, send, tl.(send\ x; send\_balances\ tl)$$

must have the type scheme

$$\forall\alpha[\alpha \leq \texttt{!clients}].$$
$$\alpha\ \mathsf{int} \to (\alpha\ \mathsf{int} \to \mathsf{unit}) \to \mathsf{client\_info\ list} \to \mathsf{unit}$$

The bank example (3)

The function `illegal_flow` tries to send information about one client to another client:

```
let illegal_flow = function
  { cash = x1 } :: { send_msg = f2 } :: _ -> f2 x1
| _ -> ()
```

Typing this piece of code yields the constraint

$$\forall \beta_1 \beta_2 . (\beta_1 \sqcup \beta_2 \leq !\texttt{clients}) \rhd (\beta_1 \leq \beta_2)$$

which is not satisfiable.

The bank example (4)

The function `total` computes the total balance of the bank from the clients file:

```
let rec total = function
   [] -> 0
| { cash = x } :: tl -> x + total tl
```

It receives the type scheme

$$\text{client\_info list} \rightarrow \texttt{!clients}\ \text{int}$$

Future work

- We intend to extend our generic type inference engine for structural subtyping, `Dalton`, in order to handle the new construct.
- Then, it will be possible to extend the `Flow Caml` system with existential and universal data-types.
- We study the possibility to make security levels also values of the Flow Caml language: this would allow to perform some dynamic tests (whose correctness must be verified statically) on existentially quantified variables when opening data-structures.

Possible work

- Giving a faithful description of the solving algorithm which describes the simplification techniques used in the implementation.
- Studying constraints resolution for other forms of subtyping.
- Introducing subtyping in more powerful extensions of ML with first order polymorphism (PolyML, ML^{F}, ...)